Magic Rescue

Relatório

Guilherme Santana, 60182
Filipe Leão, 60191

Resolução do Problema

$$V(b,i) = \begin{cases} E(b,i), & b = e \\ M(b,i), & b = 3 \vee t \vee d \\ I(b,i), & b = h \vee p \vee c \end{cases}$$

Onde E, M e I, representam as funções auxiliares Empty, Monster e Item, respetivamente.

$$E(b,i) = \begin{cases} 1 + V(b+1,\emptyset), & b = e \wedge i = \emptyset \\ min(2 + V(b+1,\emptyset), 3 + V(b+1,h)) & b = e \wedge i = h \\ min(2 + V(b+1,\emptyset), 3 + V(b+1,p)) & b = e \wedge i = p \\ min(2 + V(b+1,\emptyset), 3 + V(b+1,c)) & b = e \wedge i = c \end{cases}$$

```java
private void resolveEmpty( int[] array){
    int[] temp = array.clone();
    for (int row = 0; row < TABLE_DEPTH; row++) {
        switch (row) {
            case E -> array[row] = NO_ITEM_NO_ITEM + temp[row];
            case H -> array[row] = Math.min(NO_ITEM_ITEM + temp[E], ITEM_ITEM + temp[H]);
            case P -> array[row] = Math.min(NO_ITEM_ITEM + temp[E], ITEM_ITEM + temp[P]);
            case C -> array[row] = Math.min(NO_ITEM_ITEM + temp[E], ITEM_ITEM + temp[C]);
        }
    }
}
```

Empty

A função E(b, i) coincide com o método resolveEmpty(), da classe MagicRescue onde irá preencher uma coluna da tabela, com a resolução matemática, consoante o item.

Monster

$$M(b,i) = \begin{cases} Fill3 & b = 3 \\ FillT & b = t \\ FillD & b = d \end{cases}$$

```java
private void resolveCreature(int[] array, char pos) {
    switch (pos){
        case T_DOG -> fillTableDog(array);
        case TROLL -> fillTableTroll(array);
        case DRAGON -> fillTableDragon(array);
    }
}
```

$$Fill3(b,i) = \begin{cases} \infty & b = 3 \wedge i = \emptyset \\ 4 + V(b+1,h) & b = 3 \wedge i = h \\ 5 + V(b+1,p) & b = 3 \wedge i = p \\ 6 + V(b+1,d) & b = 3 \wedge i = c \end{cases}$$

```java
private void fillTableDog(int[] array) {
    int[] temp = array.clone();
    for(int row = 0; row < TABLE_DEPTH; row++){
        switch (row){
            case E -> array[row] = INTEGER_INFINITY;
            case H -> array[row] = T_DOG_SOM + temp[H];
            case P -> array[row] = TROLL_SOM + temp[P];
            case C -> array[row] = DRAGON_SOM + temp[C];
        }
    }
}
```

$$FillT(b,i) = \begin{cases} \infty & b = t \wedge i = \emptyset \\ \infty & b = t \wedge i = h \\ 5 + V(b+1,p) & b = t \wedge i = p \\ 6 + V(b+1,d) & b = t \wedge i = c \end{cases}$$

```java
private void fillTableTroll(int[] array) {
    int[] temp = array.clone();
    for(int row = 0; row < TABLE_DEPTH; row++){
        switch (row){
            case E, H   -> array[row] = INTEGER_INFINITY;
            case P      -> array[row] = TROLL_SOM + temp[P];
            case C      -> array[row] = DRAGON_SOM + temp[C];
        }
    }
}
```

$$FillD(b,i) = \begin{cases} \infty & b = d \wedge i = \emptyset \\ \infty & b = d \wedge i = h \\ \infty & b = d \wedge i = p \\ 6 + V(b+1,d) & b = d \wedge i = c \end{cases}$$

```java
private void fillTableDragon(int[] array) {
    int[] temp = array.clone();
    for (int row = 0; row < TABLE_DEPTH; row++) {
        switch (row) {
            case E, H, P -> array[row] = INTEGER_INFINITY;
            case C       -> array[row] = DRAGON_SOM + temp[C];
        }
    }
}
```

A função auxiliar monster é subdividida em três outras funções auxiliares - Fill3, FillT e FillD - que preenche uma coluna, de acordo com a respetiva notação matemática associada ao mostro que estiver em b, consoante o item.

Item

$$I(b,i) = \begin{cases} FillH & b = h \\ FillP & b = p \\ FillC & b = c \end{cases}$$

```java
private void resolveItem(int[][] table, int colum, char pos) {
    switch (pos){
        case HARP -> fillTableHarp(table, colum);
        case POTION -> fillTablePotion(table, colum);
        case CLOAK -> fillTableCloak(table, colum);
    }

}
```

$$FillH(b,i) = \begin{cases} min(1+V(b+1,\emptyset), 3+V(b+1,h)) & b = h \wedge i = \emptyset \\ min(2+V(b+1,\emptyset), 3+V(b+1,h)) & b = h \wedge i = h \\ min(2+V(b+1,\emptyset), 3+V(b+1,p), 3+V(b+1,h)) & b = h \wedge i = p \\ min(2+V(b+1,\emptyset), 3+V(b+1,c), 3+V(b+1,h)) & b = h \wedge i = c \end{cases}$$

```java
private void fillTableHarp(int[] array) {
    int[] temp = array.clone();
    for(int row = 0; row < TABLE_DEPTH; row++){
        switch (row){
            case E -> array[row] = Math.min(NO_ITEM_NO_ITEM + temp[E], NO_ITEM_ITEM + temp[H]);
            case H -> array[row] = Math.min(ITEM_NO_ITEM + temp[E], ITEM_ITEM + temp[H]);
            case P -> array[row] = Math.min(ITEM_NO_ITEM + temp[E], Math.min(ITEM_ITEM + temp[P], ITEM_ITEM + temp[H]));
            case C -> array[row] = Math.min(ITEM_NO_ITEM + temp[E], Math.min(ITEM_ITEM + temp[C], ITEM_ITEM + temp[H]));
        }
    }
}
```

$$FillP(b,i) = \begin{cases} min(1+V(b+1,\emptyset), 3+V(b+1,p)) & b = p \wedge i = \emptyset \\ min(2+V(b+1,\emptyset), 3+V(b+1,h), 3+V(b+1,p)) & b = p \wedge i = h \\ min(2+V(b+1,\emptyset), 3+V(b+1,p)) & b = p \wedge i = p \\ min(2+V(b+1,\emptyset), 3+V(b+1,c), 3+V(b+1,p)) & b = p \wedge i = c \end{cases}$$

```java
private void fillTablePotion(int[] array) {
    int[] temp = array.clone();
    for(int row = 0; row < TABLE_DEPTH; row++){
        switch (row){
            case E -> array[row] = Math.min(NO_ITEM_NO_ITEM + temp[E], NO_ITEM_ITEM + temp[P]);
            case H -> array[row] = Math.min(ITEM_NO_ITEM + temp[E], Math.min(ITEM_ITEM + temp[H], ITEM_ITEM + temp[P]));
            case P -> array[row] = Math.min(ITEM_NO_ITEM + temp[E], ITEM_ITEM + temp[P]);
            case C -> array[row] = Math.min(ITEM_NO_ITEM + temp[E], Math.min(ITEM_ITEM + temp[C], ITEM_ITEM + temp[P]));
        }
    }
}
```

$$FillC(b,i) = \begin{cases} min(1+V(b+1,\emptyset), 3+V(b+1,c)) & b = p \wedge i = \emptyset \\ min(2+V(b+1,\emptyset), 3+V(b+1,h), 3+V(b+1,c)) & b = p \wedge i = h \\ min(2+V(b+1,\emptyset), 3+V(b+1,p), 3+V(b+1,c)) & b = p \wedge i = p \\ min(2+V(b+1,\emptyset), 3+V(b+1,c)) & b = p \wedge i = c \end{cases}$$

```java
private void fillTableCloak(int[] array) {
    int[] temp = array.clone();
    for(int row = 0; row < TABLE_DEPTH; row++){
        switch (row){
            case E -> array[row] = Math.min(NO_ITEM_NO_ITEM + temp[E], NO_ITEM_ITEM + temp[C]);
            case H -> array[row] = Math.min(ITEM_NO_ITEM + temp[E], Math.min(ITEM_ITEM + temp[H], ITEM_ITEM + temp[C]));
            case P -> array[row] = Math.min(ITEM_NO_ITEM + temp[E], Math.min(ITEM_ITEM + temp[P], ITEM_ITEM + temp[C]));
            case C -> array[row] = Math.min(ITEM_NO_ITEM + temp[E], ITEM_ITEM + temp[C]);
        }
    }
}
```

A função auxiliar Item é subdividida em três outras funções auxiliares - FillH, FillP e FillC - que preenche uma coluna, de acordo com a respetiva notação matemática associada ao mostro que estiver em b, consoante o item.

Complexidade Temporal

Na nossa solução, recorremos ao preenchimento de um vetor com 4 posições, correspondentes ao objeto de entrada, na posição do caminho que se está a resolver.

Nesta medida, a complexidade temporal está apenas dependente do tamanho do caminho, *n, e* do número de objetos equipáveis (considerando o caso de não ter qualquer objeto equipado como mais um objeto equipável), *w,* visto que as operações de escrita e leitura de elementos do vetor e comparação entre os mesmos têm complexidades constantes.

Isto é:

$$\Theta(w \cdot n)$$

Podemos então substituir *w* pelo número de objetos equipáveis referidos no enunciado. Ora:

$$\Theta(4 \cdot n)$$

Por fim, sendo constante, podemos omitir o 4, resultando numa complexidade temporal linear. Ou seja:

$$\Theta(n)$$

Complexidade Espacial

Na nossa implementação, fazemos uso de 2 vetores cujos comprimentos correspondem ao comprimento do caminho a resolver, n, e ao número de objetos equipáveis, w. Assim, temos como complexidade espacial:

$$\Theta(w) + \Theta(n)$$

Mais uma vez, podemos substituir *w* pelo valor fornecido pelo enunciado, obtendo:

$$\Theta(4) + \Theta(n)$$

Por fim, podemos omitir a complexidade constante resultando numa complexidade espacial total de:

$$\Theta(n)$$

Conclusões

Inicialmente otimizamos uma solução onde era preenchida uma tabela de tamanho C x L onde *C* era o tamanho do percurso a percorrer e *L* era os tipos de itens que podemos carregar, incluindo o vazio. Este preenchimento era feito através de programação dinâmica. Posteriormente, de modo a melhorar a complexidade espacial, passamos a utilizar um *array*, também preenchido através de programação dinâmica, de tamanho L onde *L* são os tipos de itens que podemos carregar, incluindo o vazio. Deste modo conseguimos reduzir o espaço utilizado na nossa solução em *C*.

Depois desta alteração não encontramos possíveis melhoramentos